Chapter 6: Functions

A function can be thought of as a programmer-defined operation. Functions play a key role in programming (procedural and object-oriented). A function is uniquely defined by

- its name
- its operand types (parameters).

The actions of function are specified in a block, referred to as the function body. Every function has an associated return type.

```
#include <iostream>
using namespace std;

int gcd(int v1, int v2) { // calclate the greatest common divisor
     while (v2) {
          int temp = v2;
          v2 = v1 % v2;
          v1 = temp;
     }
     return v1;
}

int main(){
    cout << "Enter two values: \n";
    int i, j;
    cin >> i >> j;
    cout << "gcd: " << gcd(i, j) << endl;
}
```

Declaration of function first, and define it afterwards.

```
#include <iostream>

using namespace std;

int gcd(int, int); // declare first, not defined

int main(){
    cout << "Enter two values: \n";
    int i, j;
    cin >> i >> j;
    cout << "gcd: " << gcd(i, j) << endl;
}

int gcd(int v1, int v2) { // function definition
     while (v2) {
          int temp = v2;
          v2 = v1 % v2;
          v1 = temp;
```

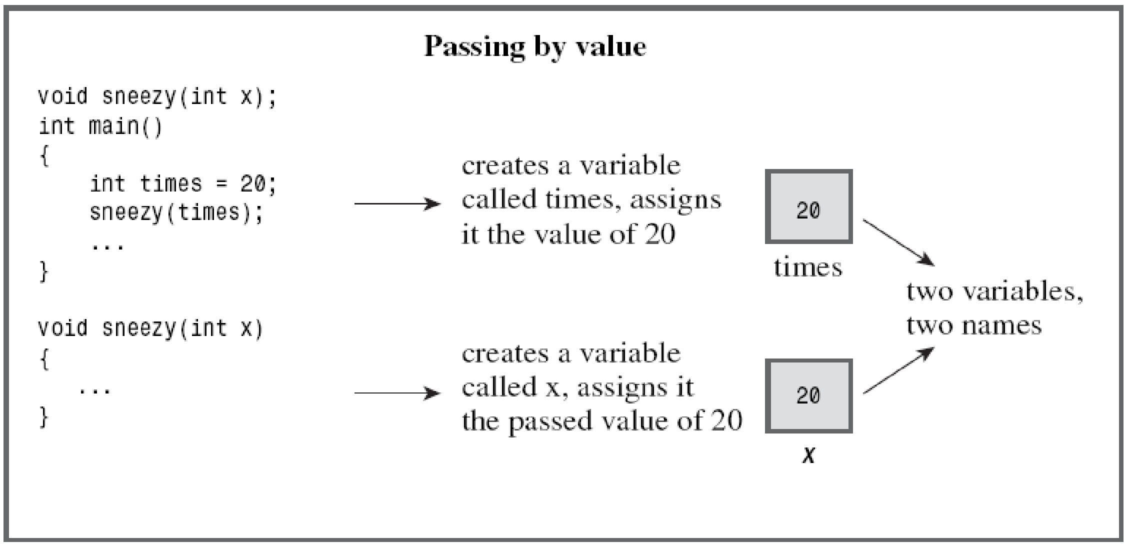
```
        }
        return v1;
    }
```

Parameters and passing arguments

When passing parameters to methods, we can:

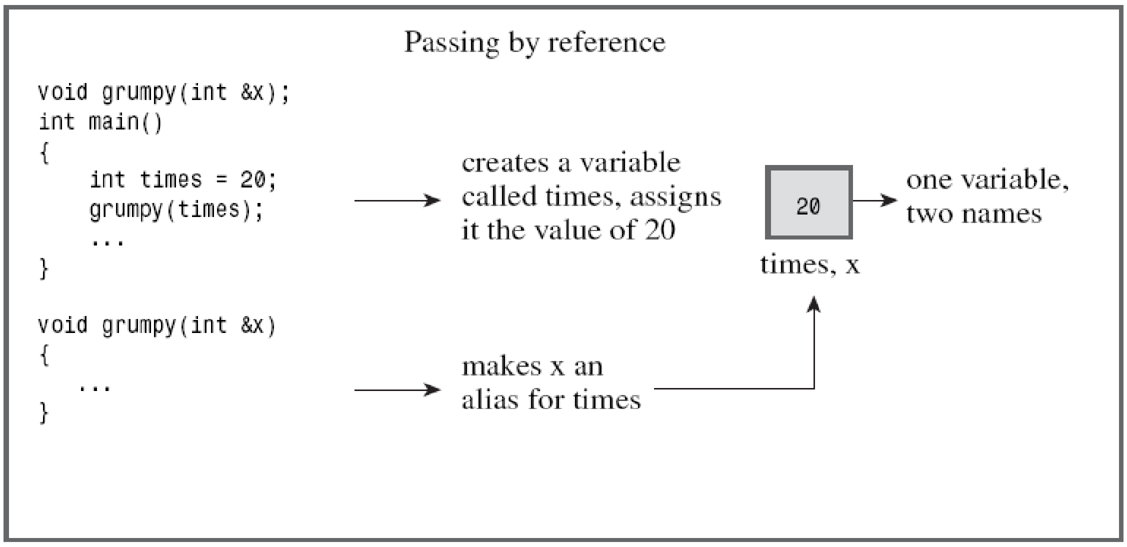
- pass nonreference and reference parameters.
- pass const reference parameters.
- pass pointer and array

Passing nonreference and reference parameters



```
void sneezy(int x);
int main()
{
    int times = 20;
    sneezy(times);
    ...
}

void sneezy(int x)
{
   ...
}
```

Passing by value
creates a variable called times, assigns it the value of 20
20
times
two variables, two names
creates a variable called x, assigns it the passed value of 20
20
x



```
void grumpy(int &x);
int main()
{
    int times = 20;
    grumpy(times);
    ...
}

void grumpy(int &x)
{
   ...
}
```

Passing by reference
creates a variable called times, assigns it the value of 20
20
one variable, two names
times, x
makes x an alias for times

```
#include <iostream>

using namespace std;

void sneezy(int);

int main(){
    int times = 20;
    sneezy(times);
    cout << "value of times: " << times << endl;  //20
}

void sneezy(int x){ // x is copy of times
    x++;
}
```

```
#include <iostream>

using namespace std;

void grumpy(int&);

int main(){
    int times = 20;
    grumpy(times);
    cout << "value of times: " << times << endl; //21
}

void grumpy(int& x){ // x is times
    x++;
}
```

Copying of memory is very time consuming. We also use reference parameters when passing a large object to a function to avoid copying. For example, large objects or large arrays.

If the only reason to make a parameter a reference is to avoid copying the argument, the parameter should be const reference. This avoids accidental modification of the parameter.

```
// compare the length of two strings
// avoid copies of strings because it could be long
bool isShorter(const string &s1, const string &s2){
    return s1.size() < s2.size();
}
```

Summary: In C++ choosing a parameter-passing mechanism is easily overlooked by the programmer that can affect correctness and efficiency. The rules are relatively simple:

- Pass by reference is required for any object that may be altered/changed by the function.

- Pass by value is appropriate for small objects that should not be altered by the function. This generally includes primitive types and also function objects.
- Pass by constant reference is appropriate for large objects that should not be altered by the function. This generally includes library containers such as vector, general class types, and even string.

6.2.4 Array Parameters

We often want to write a function to process the data in an array. An array parameter is a very special case in C++. The array name is ALWAYS followed by **an empty bracket**.

```
void fillUp(int a[], size_t size){
    // fillUp the array …
}
```

The parameter `int a[]` is an **array parameter** and `size_t` is a machine-specific unsigned type that is large enough to hold the size of any object in memory. In `int a[]`, the empty square brackets with no index expressed inside indicates an array parameter.

When passing the array as an argument, you simply pass the name of array.

```
    ...
    int a[20];
    fillUp(a, 20);
    ...
```

An array parameter to a function is the memory address of the first element (the one indexed by zero) of the array. So it **serves as a pointer.** The effect is practically pass-by-reference.

The following two lines of code are equivalent.

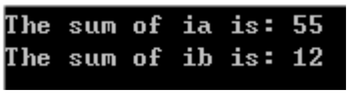
```
void fillUp1(int a[], size_t size);
void fillUp2(int* a, size_t size);
```

<u>Exercise 6.2 In-class Coding Exercise</u>

Write a `sum` function that sums up all elements in an `int` array and return the sum.

```
#include <iostream>

using namespace std;
```

```
(YOUR FUNCTION HERE)




int main(){
    int ia[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int ib[3] = {1, 3, 8};
    cout << "The sum of ia is: " << sum(ia, 10) << endl;
    cout << "The sum of ib is: " << sum(ib, 3) << endl;
    return 0;
}
```



```
The sum of ia is: 55
The sum of ib is: 12
```

Answer:

```
int sum(const int arr[], const size_t num){
    int summation = 0;
    for (size_t i = 0; i < num; ++i) summation += arr[i];
    return summation;
}
```

Because arrays are passed as pointers, functions ordinarily **DO NOT** know the size of the array they are given. They must rely on additional information provided by the caller. We explicitly pass a size. A work around is to use iterators as function arguments through `begin` and `end`:

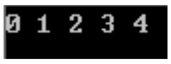
Alternative solution using `begin` and end `iterators`

```
#include <iostream>
#include <iterator>

using namespace std;

int sum(const int* beg, const int* last){
    int summation = 0;
    while (beg != last){
        summation += *beg++;
    }
    return summation;
}

int main(){
    int ia[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int ib[3] = {1, 3, 8};
    cout << "The sum of ia is: " << sum(begin(ia), end(ia)) <<
endl;
    cout << "The sum of ib is: " << sum(begin(ib), end(ib));
    return 0;
}
```

If we pass the array variable to a function, can we use `begin` and `end` in the function? No we cannot. The size of the array is not known in the function.

Recall we can use range for loop to access elements in array, `string` and `vector`.

```
#include <iostream>
using namespace std;

int main(){
 const size_t array_size = 5;
 int ia[array_size] = {0, 1, 2, 3, 4};
 for (auto i : ia)
     cout << i << " ";
 return 0;
}
```



```
0 1 2 3 4
```

However, when we pass the array into a function, the range-for **WILL NOT** work. The size of the array is not known by the function.

```
#include <iostream>
using namespace std;

void arrAutoTest(int ia[]){
    for(auto i : ia)          // this will not work
        cout << i << " ";
}

int main(){
 const size_t array_size = 5;
 int ia[array_size] = {0, 1, 2, 3, 4};
 arrAutoTest(ia);
 return 0;
}
```

Remark: for general purpose programs, you should almost always use `vector`s and iterators in preference to the lower-level arrays and pointers. Well-designed programs use arrays and pointers **only in** the internals of class implementations where speed is essential.

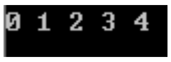
We can rewrite the above program using `vector`.

```
#include <iostream>
#include <vector>
using namespace std;
```

```
void vecAutoTest(const vector<int>& iv) {
    for (auto i : iv)
        cout << i << " ";
}

int main() {
    vector<int> iv { 0, 1, 2, 3, 4 };
    vecAutoTest(iv);
    return 0;
}
```

One way to work around the problem is to pass an array reference instead an array.

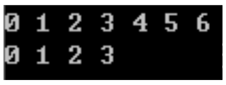
```
#include <iostream>
using namespace std;

// passing array reference to function with fixed size 5
void arrAutoTest(const int (&ia)[5]) {
    for (auto i : ia)
        cout << i << " ";
}

int main() {
    const size_t array_size = 5;
    int ia[array_size] = { 0, 1, 2, 3, 4 };
    arrAutoTest(ia);
    return 0;
}
```



```
0 1 2 3 4
```

Or a generic version:

```
#include <iostream>
using namespace std;

// passing array reference to function with generic size arr_size
template <typename T, std::size_t arr_size>
void arrAutoTest(const T (&ia)[arr_size]) {
    for (auto i : ia)
        cout << i << " ";
}

int main() {
    const size_t array_size1 = 7;
    int ia[array_size1] = { 0, 1, 2, 3, 4, 5, 6 };
    arrAutoTest(ia);
    cout << endl;
    const size_t array_size2 = 4;
    int ib[array_size2] = { 0, 1, 2, 3 };
    arrAutoTest(ib);
    return 0;
}
```



```
0 1 2 3 4 5 6
0 1 2 3
```

6.3 Return Types

- Every `return` in a function with a return type other than void must return a value.
- Return a nonreference type
  - Value returned by a function initializes a temporary (object) created at the point when the call was made.
  - Return value is copied into the temporary at the calling site
- Return a reference type
  - When a function returns a reference type, the return value is not copied. Instead, the object itself is returned.

Now let's look at a **terribly incorrect** program; **what's wrong?**

```
// this is a WRONG code

#include <string>
using namespace std;

string& manip(const string& s){
  string ret = s;
  ret += " is terrible";
  return ret;
  // warning: reference to local variable 'ret' returned
}

int main(){
  string s = "The program ";
  string sTerrible = manip(s);
  return 0;
}
```

This function will fail at run time because it returns a reference to a local object. When the function ends, the storage in which ret resides is freed. The return value refers to memory that is no longer available to the program.

Never Return a Reference to a Local Object!

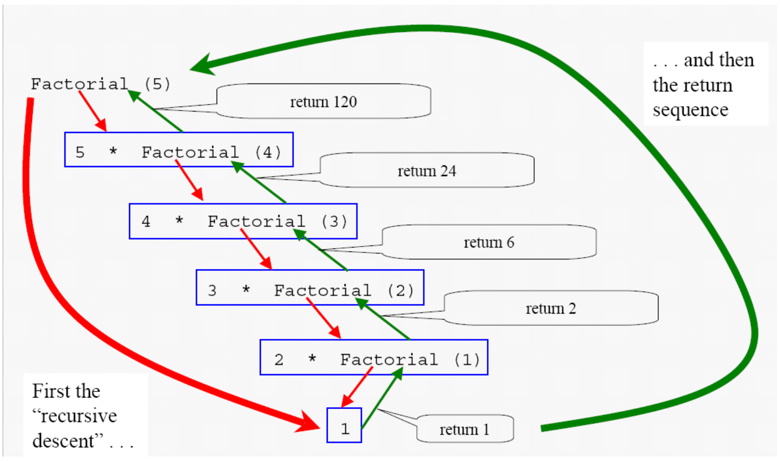
Recursive function: a function that calls itself. For example: Factorial of a number n (n!).

```
int Factorial (int val){
    if (val > 1)
```

```
        return val * Factorial(val-1);
    return 1;
}
```

Q: when will the program stop?

Q: what happens when we invoke Factorial(5)?



When finding the greatest common divisor (GCD), the following is an implementation using the loop:

```
int gcd(int v1, int v2){
    while (v2) {
        int temp = v2;
        v2 = v1 % v2;
        v1 = temp;
    }
    return v1;
}
```

A recursion version can be as follows:

```
int gcd(int v1, int v2){
    if (v2 == 0)
        return v1;
    return gcd(v2, v1 % v2);
}
```

The main:

```
int main(){
   int a, b;
   cout << "Enter the values of a and b: " << endl;
   cin >> a >> b;
   cout << "GCD of " << a << " and " << b << " is " << gcd(a, b);
   return 0;
}
```

6.4 Overloaded Functions

Functions that share the same name are said to be overloaded.

C++ compiler uses **the number of parameters** or **the types of parameters** to differentiate which version of the functions to be called.

Q: Given

```
int gcd(int v1, int v2);
```

are the following overloaded functions valid?

```
int gcd(double v1, double v2);
```

```
int gcd(int v1, int v2, int v3);
```

```
int gcd(int v1, int v3);
```

```
int gcd(double v1, int v3);
```

```
double gcd(int v1, int v3);
```

A:
Y, Y, N, Y, N

Function overloading can simplify the job of thinking up names for two or more functions that perform **different** versions of essentially the same task.

6.5.1 Functions with Default Arguments

Default arguments are the language facility in C++ that allow functions to have default values (when not provided).

```
void print(int value, int base=10){
   // some code
}
```

```
int main() {
   print(31);
   print(31, 10);
   print(31, 16);
   return 0 ;
}
```

A default argument is type checked at the time of compilation and evaluate at the time of the call. The default arguments can only be provided for tailing arguments only.

```
int f(int, int=0, int=0); //ok
int f(int=0, int=0, int); //error
int f(int=0, int, int=0); //error
int f(int, int, int=0);    //ok
```

When designing a function with default arguments, you should order the parameters so that those most likely to be used with default values appear last.

Given the following function declarations and calls, which, if any, of the calls are illegal? Which are legal but misleading?

```
void init(int ht, int wd = 80, char bckgrnd = ' ');

(a) init();        // error
(b) init(24,10);   // ok
(c) init(14, '*'); // ok
```

getline function

The C++ string class defines the **global function** `getline()` to read strings from an I/O stream. The `getline()` function reads a line from `is` and stores it into `s`. If a character *delimiter* is specified, then `getline()` use `delimiter` to decide when to stop reading data.

Syntax:

```
istream& getline(istream& is, string& s, char delimiter = '\n');
```

(Ex)

```
string s;
getline( cin, s );
cout << "You entered " << s << endl;
```

What does the above code fragment do?

A: It reads a line of text from `cin` (inputs from terminal) and displays it.

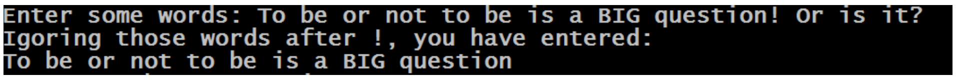
(Ex)

```
string s;
getline( cin, s, '#' );
cout << "You entered " << s << endl;
```

What does the above code fragment do?

A: It reads a line of text from `cin` (inputs from screen), stops reading when it encounters the symbol # and displays the content.

<u>Exercise 6.4 In-class Coding Exercise</u>: write a program to read some words and ignore those words after '!'. Below is a sample run:



```
Enter some words: To be or not to be is a BIG question! Or is it?
Igoring those words after !, you have entered:
To be or not to be is a BIG question
```

A:

```
#include <iostream>
#include <string>
using namespace std;

int main(){
   string sentence;
   cout << "Enter some text, terminated by !:\n";
   getline (cin, sentence, '!');
   cout << "You entered:\n";
   cout << sentence << endl;
   return 0;
}
```